CHAPTER 12

Building ActiveX, COM, and COM+ Components Fundamentals

In This Chapter

- Understanding how ActiveX fits in with COM and COM+
- Understanding how ActiveX can help you
- Determining whether you want to create an MFC- or ATL-based ActiveX control
- Understanding the problems associated with ActiveX controls
- Understanding in-process servers
- Understanding out-of-process servers
- Working with stored components
- Working with components on disk
- Understanding how ActiveX affects the registry
- Creating an ActiveDocument application

Building components is one of the major tasks that some programmers tackle. It's important to have a good source of building materials for the applications that an application needs to create. Using today's technology means creating applications using a building-block approach that allows a programmer to hook things together with glue code in a relatively short time (relative being the key word here, since any complex application is still going to take a substantial amount of time to develop). Sure, some of these building blocks are off-the-shelf components marketed by outside companies. However, there are some cases when the needs of your company are unique or at least unique enough that someone hasn't designed an off-the-shelf component to do that job yet. In this case, an in-house programmer needs to design one or more components for everyone at the company to use.

That's what this chapter is all about. We're going to explore the techniques you have at your disposal for creating both ActiveX and COM+ components. (If you don't know how COM+ can be used in place of ActiveX, you may want to read Chapter 11 first.) Obviously, the first thing you need to do is determine which technology to use. The first section does that for you. Even though both ActiveX and COM+ are based on COM, there are enough differences between the two technologies that you'll want to spend some time making a choice on which one to use.

Since ActiveX is the most common choice for component technology today, the next section explores some of the peculiarities of using this technology. For one thing, it's important to know how the various ActiveX interfaces work. We'll also explore questions such as what type of ActiveX technology to use. ActiveX encompasses more than the in-process servers that everyone automatically thinks about. There also are out-of-process server solutions such as ActiveDocument to consider. Many of your building-block needs can be taken care of by a standalone application that actually works with your application to get the job done. (For a real-world example of this technique in action, read about "The Hidden Component" in Chapter 11.) There also are two common approaches to creating the in-process server variety of ActiveX component: Microsoft Foundation Class (MFC) and ActiveX Template Library (ATL). We'll look at examples of both and discuss the relative merits of each technology.

Not all ActiveX components are created equal, even if they are based on COM. For example, you can create an ActiveX component in Visual C++ that won't work in Visual Basic because the two programming environments look for different things in a component. In fact, that's what the next section of this chapter is all about. We'll look at the gotchas that Visual C++ programmers have to look out for when creating components that need to work in both Visual Basic and Visual C++. Obviously, a lot of the things discussed here also affect a developer who works in environments other than Visual C++ or Visual Basic. For example, you'll find that Visual InterDev closely resembles Visual Basic when it comes to displaying the ActiveX control and presenting the properties that it provides.

The final section of the chapter looks at the main out-of-processor form of ActiveX, ActiveDocument. Examples of common ActiveDocument applications include most of the applications in Microsoft Office. (We'll concentrate on Word 97 specifically, but the same principles apply to the other Office 97 products.) The user can use an ActiveDocument application to view documents online without leaving the browser environment, which is a real plus when you want your users to access company documents through the company intranet instead of searching the network by hand.

As part of the ActiveDocument discussion, we'll look at the differences between in-process and out-of-process servers. An in-process server uses the same memory and session as the host application, whereas an out-of-process server executes in its own memory space and session. This section of the chapter explores what you need to know about these two server types and when you should use them. We'll also talk about the various advantages and disadvantage of using each server type. In some cases, you'll find that there really is only one choice to make when it comes to choosing between in-process or out-of-process servers.

ActiveX Versus COM+

There are two technologies that you'll run into on a regular basis when dealing with components: ActiveX and COM+. Both technologies are based on COM, as discussed in the previous chapter. However, there are enough differences in the implementation details of these two technologies that you may need to make decisions based on which technology is best for you to use at a given time.

Most components today are created using ActiveX. That's because ActiveX has quite a few things going for it that you can't get with COM right this second (although Microsoft certainly will do its best to overcome these shortcomings as time progresses). The following list provides some ideas of why this technology may be the answer for you right now.

- **Support:** ActiveX is the predominant technology right now and probably will be for quite some time to come. There are a lot of books and other resources available for ActiveX that you may have a hard time finding for COM+ components. In addition, there's a wealth of ActiveX-specific Web sites and newsgroups on the Internet. As of this writing, there aren't any COM+ specific newsgroups, although Microsoft probably will have one in place by the time you read this.
- **Reliability:** ActiveX is an established technology, whereas COM+ is still a moving target in some ways. As a result, ActiveX offers a level of stability that COM+ can't provide right now. In addition, because there are more ActiveX controls on the market, most of the interoperability and other programmer related issues are known and well documented. Since COM+ is a relatively new technology, it will take time for programmers to document any potential problems.
- **Diversity:** You can choose from more than one flavor of ActiveX. When it comes to creating in-process components, there are two prevailing strategies: MFC and ATL. However, you aren't limited to these two strategies. You can also create ActiveX components using the BaseCtrl framework example in the ActiveX SDK as a template or completely by hand. Obviously, these latter two choices require you to have in-depth knowledge of COM, which is why they're less popular than the wizard derived alternatives discussed later in this chapter.

Obviously, most of these issues are time related. For example, as developers learn more about COM+, it will become better documented and you'll be able to get the same level of support for it as you can for ActiveX. In addition, COM+ will likely be more reliable than ActiveX simply because of the way that it's designed. Unlike ActiveX, you should run into few, if any, language-related problems with COM+ because of its self-discovery feature. (For a full discussion of the new features that COM+ will provide, read the COM+ section of Chapter 11.) In sum, though ActiveX is probably the best choice today when you need an established technology, it's not too soon to begin using COM+ for other types of programming needs.

Why ActiveX?

ActiveX represents one of the most flexible methods for creating components today — at least if you look at all of the ways that ActiveX is used. Not only can you create windowed ActiveX components for in-process use within a desktop application, but there also are nonwindowed versions for server use. In addition, ActiveX is attached to a variety of other component forms, including applications that can be used in an out-of-process mode.

The following sections discuss ActiveX in depth. Let's begin by taking a better look at what ActiveX can do for you. Next, we'll look at when you should use ActiveX and what form of ActiveX component is best for a given situation. Finally, we'll look at what you need to know about ActiveX from a programming perspective by discussing the various interfaces that you need to implement.

What Will ActiveX Do for You?

ActiveX has done for the Internet what OCXs did for the desktop. Component technology has become part of many Web sites and the number grows every day (albeit somewhat slowly). The ATL-based form of the ActiveX control is small, fast, and extremely flexible. In addition, with the improvements in the ATL wizards for Visual C++ 6.0, you'll find that development isn't nearly as slow as previous versions. (The MFC development cycle remains an order of magnitude faster and still represents the better development method to use for desktop application components.)

You'll find ActiveX controls in places that you may not have thought about using component technology in the past. For example, NetManage, Inc. has created a UNIX e-mail client called *Z-Mail*. This package supports ActiveX technology in a way that allows users to exchange, create, and view HTML documents directly in the message-viewing window. What this means to users is that they now have the ability to create dynamic Web pages.

You can learn more about Z-Mail (and other NetManage products) at `http://www.netmanage.com/products/zmail/`. This Web site also contains a wealth of white papers that are worthwhile reading since many of them talk about collaborative efforts between NetManage and Microsoft.

Remote connections will also benefit from the use of ActiveX. For example, Proginet Corporation has created an ActiveX technology that will bring mainframe data to the desktop. Their *Fusion FTMS (File Transfer Management System)* will work with any development language that supports OLE containers, such as Delphi, Visual C++, and PowerBuilder. Essentially, you'll place an ActiveX control on a form, define where to find the data, and then rely on the control to make the connection. No longer will remote access over the Internet require the user to jump through hoops.

A special transfer server on the mainframe will complete the package by automating all transfer requests. No longer will an operator have to download a needed file manually to the company's Web site before a client can access it.

You can learn more about Fusion FTMS at `http://www.proginet.com/Products/fusion/fusionft.htm`.

Even Microsoft Exchange has benefited from ActiveX. Wang Laboratories, Inc. and other companies are creating new add-ons that mix Exchange and ActiveX together to enhance Exchange capabilities. Wang's product is a client/server imaging add-on. It will allow users to scan, view, annotate, manipulate, or print graphic images no matter where they are located. This same product will include a hierarchical storage management ActiveX control. The two technologies will work together to make graphics easier to access and use in a large company. They'll also make it easier to find a needed graphic — which should ultimately result in storage savings to the company.

Microsoft itself has issued a whole slew of ActiveX controls. Some of these controls are free for downloading from their Internet site (`http://www.microsoft.com`). Examples of these new controls include Animation Player for PowerPoint and Internet Assistant for both Access and Schedule+. The Internet Assistant for Access will create a snapshot of a database table that gets uploaded as a static image. The snapshot automatically updates every time the user accesses the page. The Internet Assistant for Schedule+ will allow you to upload scheduling information to a Web page. Since the data gets updated every time a user accesses the site, you no longer have to worry about compute-at-home employees missing meetings. Finally, the PowerPoint Animation Player will allow you to play a PowerPoint presentation from within any ActiveX-compliant browser.

In addition to their own components, Microsoft is always on the lookout for new components to tell the development community about. The following list provides you with some ideas of where to look on Microsoft's Web site for additional ActiveX component information.

- `http://premium.microsoft.com/msdn/library/download/download.htm`: This Web site usually gets updated every other week with fairly new ActiveX components that will make your life easier. In many cases, you can download a free evaluation version of the component right from the Microsoft Web site. You'll also find a complete description of the ActiveX component and some information about the company that developed it. (Unfortunately, you normally have to go to the sponsoring company's Web site to get telephone and address information for the featured ActiveX component.)
- `http://msdn.microsoft.com/vstudio/downloads/resguide/p35.asp`: This Web site provides some additional components that you can download and evaluate. At the time of this writing, the list contains only one entry, but this probably will change as Microsoft develops the site.

- `http://www.microsoft.com/officedev/index/activex.htm`: If you're more interested in learning about ActiveX components that may already appear on your machine, then this is the place to check out. This Web site talks about the ActiveX components used by Microsoft Office products and could provide you with valuable information to reduce your own component development costs.
- `http://www.microsoft.com/workshop/author/htmlhelp/actx.asp`: For those of you creating a Web-based help page, this is the place to look. The HTML Help ActiveX control can help you create things such as a table of contents for your Web site. This Web site also provides a sample that demonstrates the HTML Help control in action (the sample contains a help file that shows how to use the HTML Help file). Any machine that has Internet Explorer 4 installed, also has this control at its disposal, so you don't need to download anything to start using the technology.

Finally, if you think ActiveX doesn't help with security, think again. A lot of new firewall and certificate strategies are making the rounds these days. One of them is Net2000. It's a set of APIs that will allow developers to tie NetWare core services (including directory, security, and licensing) into their applications. You'll be able to tap this API through ActiveX controls over an intranet. How will this help users and developers alike? It means that with the proper programming constructs, a network administrator can track license usage throughout the entire network, even across Internet connections. This will become a much bigger issue as more people begin to compute from home rather than the office.

Deciding When to Use an ActiveX Control

As previously mentioned, ActiveX isn't always the technology of choice on the desktop or the Internet. Consider the desktop for a moment. There are times when you may want an out-of-process server solution such as ActiveDocument because you need to work with documents instead of augmenting a specific application feature. Even though you can create MFC-based ActiveX controls that can do anything that a desktop application can do, there are times when the memory handling and processing capabilities of an out-of-processor is actually a better solution. An easy way to figure out which technology is better is to determine the worst-case scenario if the server ran out of memory or crashed. More often than not, you can recover from an out-of-process server crash.

Microsoft would like you to believe that ATL-based ActiveX controls are one of the best things to ever happen to the Internet. After all, ATL-based ActiveX controls are much smaller than their MFC counterparts — and run faster as well. However, this isn't always true. There are situations where another form of object technology such as a Java applet may be the best choice because you need cross-platform compatibility.

In short, one of the more important things you can determine before you even begin an ActiveX control project is whether it's even the right choice for you. The

following sections answer two questions concerning ActiveX control usage. First, we'll look at how ActiveX compares to an older technology such as OCX. Second, we'll compare ActiveX control development environments: MFC versus ATL.

There are a lot of ways to create an ActiveX control other than using Visual C++. Both Visual Basic and Visual J++ will allow you to create ActiveX controls. There also are ways to create an ActiveX control using a scripting solution such as a VBScript scriptlet. In fact, there are times when using one of these languages may be a plus, especially if you don't know how to use Visual C++ and development time is a factor. The main reasons for using Visual C++ to create your ActiveX controls are the size of the final control (Visual C++ produces the smallest controls), speed of execution (Visual C++ also produces the fastest controls), and packaging (the ATL version of ActiveX controls are self-contained). In addition, you can't protect your code investment when using a solution such as a VBScript scriptlet or a Java applet. Both Visual Basic and Visual C++ help you protect your investment by using compiled code for the component.

ActiveX versus OCX Controls

For the most part, ActiveX and OCX controls are totally interchangeable. You'll see ads for ActiveX controls that have nothing to do with the Internet. Look a little closer and you'll find that those controls probably appeared in an OCX listing sometime in the not-too-distant past. Of course, you have to watch these ads carefully. An ActiveX control isn't the same thing as an OCX — even if they do share the same heritage. Remember that an ActiveX control has to be able to work with the Internet — it must support both incremental rendering and asynchronous connections.

There are some practical needs to consider as well, especially when an ActiveX control is used on the Internet. The Internet places some special challenges on the programming environment. For one thing, you don't have the luxury of high-speed loading anymore, so the size of an OCX version of a component becomes a critical issue when used within the Internet environment. Since ActiveX controls are 50 to 75 percent smaller than an OCX control of the same functionality, they work better for Internet use than an OCX control. Unfortunately, even a 75 percent decrease in size won't allow some ActiveX controls to work well in an Internet situation — you still have to consider the download time for the component from the user's perspective.

Some expectations for ActiveX support on the Internet are unrealistic even if you do use ATL-based ActiveX controls to do the work. Even if an ActiveX control is available for performing a certain task and the user can expect to download the control from your server in a reasonable amount of time, the data transfer rate determines whether the ActiveX control will work as anticipated. In other words, don't expect PowerPoint or other graphics-intensive presentations to work well over a 33.6Kbps connection, even if you have a great ActiveX control with which to play it. Make sure that you take the data transfer rate and capabilities of the client machine into account before using an ActiveX control to accomplish a given task.

There are some problems when working with ActiveX controls that you'd never need to consider (and still may not need to consider if your control is of such size that use on the Internet is impractical). ActiveX controls that can be used in a variety of environments suffer from various machine-specific requirements. When you install an OCX or a desktop-only ActiveX control on a machine, the installation program can check the machine and make allowances as needed. The same can't be said about an ActiveX control that can function on the Internet. You can't assume anything about the client machine at all. It can be anything from a new Pentium to yesterday's 80386. (If your ActiveX control does have some kind of platform limitation, you either have to find some way to work around it or make certain that everyone who uses it knows about the limitation.)

You also need to deal with some situations that an OCX programmer never would have to think about. For one thing, what happens if the browser doesn't support ActiveX at all?—which is a problem with Netscape product users. The current method of dealing with this is that the browser simply ignores any HTML tags that it doesn't know how to work with. Dealing with a browser that's not ActiveX-compliant is easy in this case—just leave a message telling the user that his or her browser won't work with the current page, and direct the user to an alternative. You could also use an ISAPI solution such as the one I talked about in Chapter 8 that relies on a script to direct the user to the right Web page.

MFC versus ATL (ActiveX Template Library)-based Controls

Visual C++ shows its robust development environment when compared to other programming languages by including two ways to create ActiveX components: MFC or ATL. In fact, the ATL-style controls support a wide variety of both windowed and nonwindowed component options that you can use to create client-side and server-side components. Both methods offer a wide variety of base object types, although the ATL-based controls may have more base object options to choose from and those options may be presented in a clearer way. (The base object type allows you to develop components faster through the process of aggregation—in short, you borrow the features of the base object, then add the features you want the resulting component to have.)

Unfortunately, the development flexibility offered by Visual C++ can cause some problems that other developers don't have to face. For one thing, how do you determine which type of control to create? Some developers have rendered the question moot by using the same technology for all of their controls. But following this route means that you haven't really explored and used Visual C++ to its full potential.

There really isn't any way to say definitively that one method of creating a control is better in a given situation. What you really need to do is define what you expect the control to do and what you're willing to invest to get that functionality, as well as your level of expertise. Obviously, there are situations where one control

creation method is preferred over another because the two methods do have distinctly different advantages and disadvantages. To give you some idea of what you need to consider when looking at an ATL ActiveX control versus one created using MFC, read through the following list. You can use these ideas to help you make a decision on which route is best for you.

Development speed Using the MFC ActiveX Control Wizard is the fastest method to create a control. The wizard takes care of most of the interface details so that what you end up with is a skeleton that's ready for some control logic. In fact, it usually takes the developers twice as long to use the ATL method for creating a control. Obviously, your results will vary depending on factors such as control complexity and your programming experience.

Control size If you want to create the smallest possible ActiveX control with wizard support, then go the ATL route. The ATL-based controls don't have to carry around any MFC baggage, nor do they rely on external MFC libraries, which makes the resulting control much smaller.

If you want to get the most efficient control from a size perspective, you can use the BaseCtrl framework, but this requires a programmer with advanced COM experience and a lot more development time than any of the wizard-oriented methods discussed in this chapter.

Development flexibility ATL gives you full control over every aspect of the control and makes it feasible for you to hand-tune every control element without getting bogged down in MFC-specific code. Not only are MFC-based controls larger, but also remember that clients may have to download the MFC libraries before they can use the control, which is a significant amount of code.

Learning curve ATL controls are much harder to create than MFC controls simply because you have to consider more things such as exactly how the interfaces will work. In most cases, it pays to create your first couple of controls using the MFC ActiveX Control Wizard so that you can learn the ropes of creating the control logic.

If you're used to the Visual C++ Version 5.0 ATL support, try the new ATL COM AppWizard in Visual C++ Version 6.0. The new ATL COM AppWizard helps you create controls much faster than the manual techniques of the past. In addition, inserting objects is much easier than before and you'll find yourself spending less time monitoring all of the interface details. However, using the MFC ActiveX Control Wizard is still far faster than using the ATL COM AppWizard to create a control. You'll find that there's a definite difference in the two control creation methods from a development-time perspective.

Compatibility — By definition, MFC-based ActiveX controls require the client to have the MFC libraries installed on their machines. However, there are more than a few versions of those libraries floating around and they aren't all compatible. What happens when users download your control and the associated libraries then can't use an important application because the new libraries are incompatible with their applications? Since the MFC libraries are stored in the SYSTEM directory, a client machine can have only one version. That's where the compatibility problems come into play.

Ease of use — The MFC ActiveX Control Wizard tends to throw everything but the kitchen sink into a control because it assumes nothing about your ability to write control code. What this means is that you end up with a wealth of interfaces you may not need or use. All of this wasted functionality bloats the size of the control and makes it harder to use.

Ease of code modification — Creating an MFC-based control is very easy the first time around. Since the wizard adds much of the code for your application, development goes quickly. However, what happens when you decide to update that control? Now you have source files that may contain a good deal of code that wasn't written by a staff programmer and that may require additional time to research and understand.

Understanding the Required Interfaces

All COM objects use interfaces. At its core, an interface is nothing more than an array of pointers to the methods provided by your ActiveX control or other COM

components. An interface provides a reliable, language-independent method for accessing the methods and properties within an object. In fact, some programmers have been known to describe a simple ActiveX control as a COM component that implements the IUnknown interface. However, to be useful, an ActiveX control has to implement more than the IUnknown interface.

You can use interfaces to provide every piece of information about your ActiveX control and how it operates. Using interfaces allows anyone to use an ActiveX control and ensures that the control itself remains safe from the vagaries of data corruption. Of course, this assumes that the interface is created correctly and the control adheres to Microsoft specifications.

I discussed most of the important general facts about interfaces throughout Chapter 11, so I won't cover all of the technical details about interfaces in general again here. However, it's important to understand how interfaces affect ActiveX components in specific. The following sections of this chapter talk about ActiveX component interfaces and how various design choices you make affect the kinds of interfaces that a component contains.

It's very important to remember that all forms of COM rely heavily on the registry. As a result, you may see component problems that have nothing to do with your code, but with faulty registry entries. Make sure you look for every potential cause of errors when debugging an errant component. Though the various Visual C++ wizards do a terrific job of creating the registry entries required to support your component, there are times when it pays to take a look at what entries were made and why. Chapter 5 has full details about the contents of the registry and what you can expect when looking for various types of registry entries. Pay special attention to the CLSID key description because this is where your component entries will appear in the registry.

Understanding the ActiveX interfaces

Interfaces encompass the idea of standardized access, which is the main goal of using component technology to build applications. Not only do components allow you to reuse code, but they also allow you to do so using standard methods that don't rely on an in-depth knowledge of the inner functioning of the component itself.

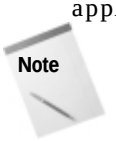
Microsoft publishes standards that enable everyone who creates an ActiveX control to create something that everyone can use. These standards enforce the idea that the interface follows a specific set of rules when providing information about the control that it supports. So, at a higher level, an interface is also a standard — an ideology about how things should work.

No matter how you view interfaces, they're a fact of life for anyone who creates ActiveX controls. You must create specific interfaces that do certain things for the control to work properly. However, there's another part of the picture to consider.

Remember that an interface defines the relationship between the client and the server, so both the client and the server have something to say about the interface requirements.

Visual C++ supports a variety of ActiveX control types, everything from full control to one that's optimized for working with Internet Explorer alone. The interfaces you need to provide for the component to work with a specific environment depends a great deal on the environment itself. In other words, a component designed for Internet Explorer use may have different needs from one designed for use with applications in general.

With the client, server, and environmental needs in mind, we'll look not only at required interfaces in this section, but also at the interfaces required to work in a specific environment. Table 12-1 provides a comparison of various ActiveX control environments and the interfaces they require. Note the special I<Class> entry in the list. This particular interface differs for every component because it contains the elements specific to that component, including both methods and properties. Obviously, this table shows the minimum number of interfaces; you could easily extend the I<Class> entry to include several interfaces if needed by your particular application.



The Composite and Lite Composite ActiveX control environments are based on CComCompositeControl class instead of CComControl class. The CComCompositeControl class allows the control to contain other Windows controls and to host multiple controls in a single control. In addition, the HMTL and Lite HTML ActiveX control environments implement two IDispatch interfaces. The first interface interacts with the control itself. The second interface handles user interface events, methods, and properties. The second interface also includes one default method, OnClick( ).

Table 12-1
Interface Support Required for Various ActiveX Control Types

Interface	*Full*	*Lite*	*Composite*	*HTML*	*Lite Composite*	*Lite HTML*
I<Class>	X	X	X	X	X	X
IDispatch	X	X	X	X	X	X
IPersistStreamInit	X	X	X	X	X	X
IOleControl	X	X	X	X	X	X
IOleObject	X	X	X	X	X	X

Interface	*Full*	*Lite*	*Composite*	*HTML*	*Lite Composite*	*Lite HTML*
IOleInPlaceActivateObject	X	X	X	X	X	X
IViewObjectEx	X	X	X	X	X	X
IViewObject2	X	X	X	X	X	X
IViewObject	X	X	X	X	X	X
IOleInPlaceObjectWindowless	X	X	X	X	X	X
IPersistStorage	X		X	X		
IQuickActivate	X		X	X		
IDataObject	X		X	X		
IProvideClassInfo	X		X	X		
IProvideClassInfo2	X		X	X		

There are many other interfaces that ActiveX controls have to support to obtain certain levels of functionality. For example, if you want to support events in your ActiveX control, then you need to provide an IConnectionPointContainer interface. There are some cases when you need to provide interfaces in addition to the one that you want to support. For example, the IConnectionPointContainer interface also requires you to implement the IEnumConnectionPoints, IConnectionPoint, and IEnumConnections interfaces. Another common supplementary interface is ISpecifyPropertyPages, which tells the container application that your control includes at least one property page. The property page itself needs to support the IPropertyPage or IPropertyPage2 interface.

Using the ActiveX interfaces

In the previous section, I talked about the interfaces you were required to provide given a specific ActiveX control environment. Now that you have a little better idea of which interfaces are required where, let's talk a little about the purpose of those interfaces. The following list provides a brief description of each major ActiveX control interface.

IDataObject

This interface defines everything needed to transfer information between two objects, including the data format and the method used to transfer the information. If the data can appear in more than one format, this interface provides an enumerated list of available formats. This also is the interface that provides information about data rendered for a specific device (making it unsuitable for devices that don't conform to a specific standard).

IDispatch

There are two methods for accessing the methods and properties provided by an ActiveX control. IDispatch provides an indirect method that relies on a type library for implementation purposes. This interface is always used as part of a dual-interface control. A dual interface isn't necessarily a requirement by Microsoft standards. However, most developers today implement a dual interface to allow their control to work with more than just the language in which it was developed. IDispatch is a late-bound interface.

IPersistStreamInit

Making sure that a control's data remains intact between uses is important. (Otherwise, you have to reconfigure your applications every time you use them.) Use this interface to make sure that any stream data required by your control is persistent or saved between sessions. This replaces the older IPersistStream interface. The main difference between the two interfaces is that IPersistStreamInit also includes initialization code in the form of the InitNew( ) method, which initializes a control to a default state.

IOleControl

Communication is key with ActiveX controls. The overall purpose of this interface is to control communication between the client and server. The normal purpose of this interface is to provide keyboard information. For example, a client can ask about the control's keyboard behavior and whether it provides full or partial support for control key combinations.

IOleObject

This is the most essential of all interfaces for an ActiveX control, since it provides both basic control functionality and communication features. Along with this interface, a control must provide support for both IDataObject and IPersistStorage for every embedded object that it supports. IOleObject contains a wide variety of methods designed to enhance control functionality. There are a minimum of 21 interface-specific methods along with the three standard methods (QueryInterface( ), AddRef( ), and Release( )) for a total of 24 methods in a standard IOleObject interface. Of the 21 interface-specific methods, you can ignore six by returning E_NOTIMPL (error not implemented) if you don't require the functionality they provide: SetExtent( ), InitFromData( ), GetClipboardData( ), SetColorScheme( ), SetMoniker( ), and GetMoniker( ). Three of the methods — DoVerb( ), Close( ), and SetHostNames( ) — require a full control specific implementation.

IOleInPlaceActivateObject

Some types of ActiveX controls require a method for communicating with the container frame and documents. Normally, this interface is used when a control needs to support MDI or other complex application environments. This particular

interface allows visual editing of the control in certain application types like a programming language IDE.

IViewObjectEx, IViewObject2, and IViewObject

All three of these interfaces have one thing in common: They allow the ActiveX control a certain measure of autonomy in displaying itself without passing a data object to the container. Not only does this speed up the drawing of the control, it ensures a minimum of drawing problems. The caller can request specific types of drawing features.

For example, the ActiveX control can choose between a full or iconic display of the object. The IViewObject2 improves on IViewObject by returning the size of the drawing required to represent the object when using a specific presentation. You normally use this interface when working with compound document containers. The IViewObjectEx interface includes all of the features of IViewObject2. It improves on IViewObject2 by adding flicker-free drawing for nonrectangular objects, hit testing for nonrectangular objects, and control sizing. Use this interface to implement irregularly shaped controls.

IOleInPlaceObjectWindowless

Use this interface to allow a windowless control to receive window messages and to participate in drag-and-drop operations.

IPersistStorage

Storing your data objects from one session to the next is an important part of making the control react the same way each time you use it. This interface provides a structured storage medium where each object has its own storage area within the container's storage medium. You must implement this interface along with IOleObject and IDataObject to make the control work within an embedded environment.

IQuickActivate

Performance is often a factor when working with ActiveX controls, especially if the user has already spent time waiting for the control to download from the Internet. This interface allows the control and container to combine load-time or initialization-time handshaking into a single call, greatly improving control performance.

IDataObject

This interface works with the control's data. It allows the container and client to exchange information and also provides the means for one object to notify the other of data changes. The data-transfer methods supported by this interface allow objects to specify the format of the data or enumerate the available data formats. The client can also specify that data is rendered for a specific device.

IProvideClassInfo and IProvideClassInfo2

Use this interface to access a control's coclass entry in its type library. The IProvideClassInfo2 interface simply is a faster version of the IProvideClassInfo interface. It also provides an ancillary method, GetGUID, which returns the object's outgoing IID for its default event set.

ActiveX Gotchas for Visual C++ and Visual Basic Programmers

I've spent a lot of time talking about the theory behind writing and testing component code in this chapter, so you should have a good understanding of the complexity of creating even a simple control that provides a fully functional interface. (I show you how to create several kinds of ActiveX controls in the next chapter.) However, even though we've covered a lot of detail about the creation of components, we haven't spent a lot of time looking at the problems you might encounter when working with more than one programming language.

Creating a component that works in every environment may seem like a simple task given that Microsoft has put together a specification that everyone should follow, but it's not as easy as you might think. Even Microsoft hasn't followed the COM standard completely. For example, Visual Basic treats some types of nonwindowed controls differently than Visual C++ does, whereas environments such as Internet Explorer expect something different. In essence, you have to know in advance what interfaces an environment is looking for and how they expect that interface to be implemented before you know that the control will work in all the environments you designed it for.

Consider a simple timer control for a moment. When you design this control in Visual C++, you more than likely make the control invisible at runtime. However, choosing this option within the MFC ActiveX ControlWizard adds a OLEMISC_INVISIBLEATRUNTIME flag to your code. This flag, in turn, tells the client that the control wants to be activated without a window. Although this methodology works just fine within Visual C++, it doesn't work at all within Visual Basic. What Visual Basic does is ignore the control—it doesn't in-place activate an invisible control. To fix this problem, you need to make the control visible, but reduce its window size to 0. In essence, the user can't see the window that you need to provide to make the control visible within Visual Basic.

Even something as small as the name of your component is treated differently in Visual Basic than it is in Visual C++. When I set up the ActiveX controls in this chapter, I typed a long name in the Type field of the wizard used to create the control. As you saw from the various test setups, Visual C++ and its associated utility programs used this long name when presenting the control on screen.

However, if you open Visual Basic, you'll find an entirely different name when you look through the component list. To display the same name in both Visual C++ and Visual Basic, you need to change the IDispatch helpstring value as shown here in bold:

```
[
    uuid(1595BAB2-B60D-11D2-B948-00A02456674B),
    version(1.0),
    helpstring("Multiple State Pushbutton Control (ATL)")
]
```

COM+ is supposed to fix a lot of these problems because the interface isn't coded in stone. The control becomes self-describing. In essence, it provides a more flexible description that every programming language can understand. Obviously, at the time of this writing, that ideal situation lies in the future. For today, you still need to test your control in every environment that you intend to use it in before you can say that it will work as expected.

Working with ActiveDocument

ActiveDocument is a newer technology designed to work in the Internet environment, specifically with Internet Explorer (although you can create your own custom browser that displays ActiveDocument documents). The feature that makes ActiveDocument so different is that it allows you to display standard documents within a browser window, even if the browser doesn't support that document type.

To see an example of ActiveDocument at work, create a simple Web page with a link to a Word document on it. Click the link and you'll see the document opened in the browser window. Figure 12-1 shows an example of a Word document displayed in Internet Explorer. Notice that Word has taken control of the Internet Explorer toolbar, menu, and other control elements—just as it would if you embedded the document within another application such as Excel.

In reality, ActiveDocument is just another form of COM. In this case, however, you usually deal with an entire application (or the document/view subset) instead of a simple component. In addition, ActiveDocument is an out-of-process server, versus an in-process server like ActiveX components are for the most part.

This section of the chapter helps you understand ActiveDocument. Interestingly enough, the same programming techniques that I've shown you for components work just fine for ActiveDocument applications. The main difference is in the interfaces that an ActiveDocument application implements.

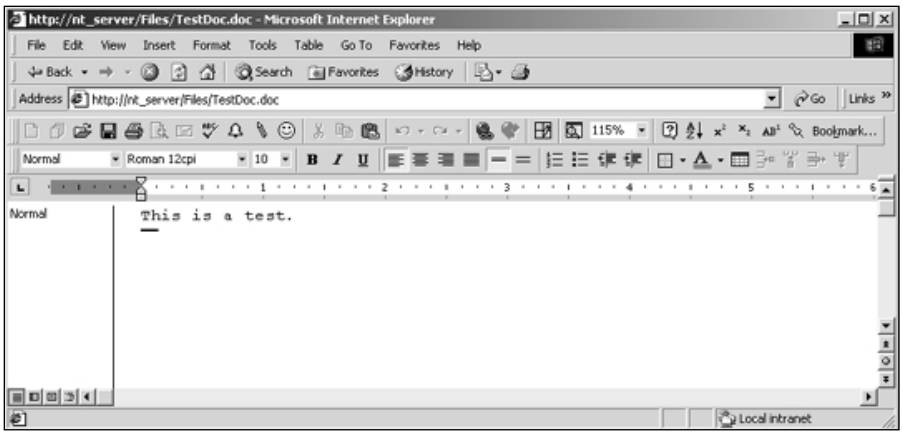


Figure 12-1: Word is just one example of ActiveDocument in action.

Out-of-Process versus In-Process Servers

COM objects can be grouped by the way they load. Either the server loads in the same memory and address space as the client application, or it loads outside of that client as a separate process with its own memory. That's the most basic difference between in-process and out-of-process servers — where and how they load.

Obviously, this isn't the end of the story. In-process servers are normally contained in DLL files — so some developers call them DLL servers. It's important to remember that OCX files essentially are renamed DLL files in this case. On the other hand, since out-of-process servers are normally standalone applications, they reside in EXE files and are called EXE servers by some developers.

It's easy to see how an in-process server and a container communicate. After all, they reside in the same address space, which makes passing data back and forth relatively easy. All that the container needs is an address for the required interface and a list of methods to work with. However, the operating system protects the out-of-process server from direct access by another process. In other words, any address that the container might get from the server is useless.

An out-of-process server relies on a proxy/stub mechanism to accomplish its work. The proxy gets installed in the client's memory space. As far as the client is concerned, it still talks with a local server. In reality though, it's talking with the server's proxy. Likewise, the stub gets installed in the server's memory space. As far as the server is concerned, the container is installed in local memory. The operating system takes care of any communication required to transfer information from the proxy to the server and from the stub to the container. Since the proxy/stub mechanism is fixed, the data transfer can take place safely.

All in-process server calls are local. What this means to the programmer is that you don't need to take some potential out-of-process server problems into account.

There are actually two forms of out-of-process server calls: *Local Procedure Call (LPC)* and *Remote Procedure Call (RPC)*. Looked at simply, the difference between the two is that an LPC occurs on a local machine, whereas an RPC occurs on a remote machine. When a container calls on the services of an out-of-process server that requires the use of RPC, it can run into myriad problems, including disconnections and delays. I discuss how to handle these problems more in Chapter 13, but for now all you need to know is that an in-process server is normally easier to implement simply because of the kind of communication it provides.

Microsoft didn't create the RPC specification. It was created by the Open Software Foundation. You can find out more about the Distributed Computing Environment (DCE) RPC specification at `http://www.osf.org/` or `http://web1.osf.org/`. There's also a good overview of how RPC works at `http://www.ja.net/documents/NetworkNews/Issue44/RPC.html`.

Both out-of-process and in-process servers must implement the IMarshall interface, which provides the means for moving data between container and server. Normally, this requirement is taken care of for you through the MIDL compiler that works with the IDL/ODL files discussed throughout this chapter. In other words, the description of the interfaces for your COM server (ActiveX component throughout most of the chapter) is enough to provide an IMarshall implementation.

Obviously, implementing IMarshall for an in-process server is a lot easier than for an out-of-process server because of the memory issues involved. Though it isn't hard to move an integer from the server to a container, what do you do about structures? Since the memory pointed at by the structure isn't accessible to the container, the entire structure has to be copied, which can present problems. Now, what happens if the server passes a pointer to a structure? It's no longer a matter of copying the structure from the server to the container; the marshalling mechanism can only pass the pointer provided by the server, then use that pointer to allow the container to gain access to data stored in the server memory space as needed.

As you can see, the whole topic of marshalling can become quite complex, especially when it comes to an out-of-process server implementation. Fortunately, the default implementation of the IMarshall interface is enough — the only time you need to override it is for performance considerations.

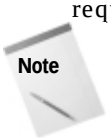
ActiveDocument Required Interfaces

There actually is an easy way of looking at the OLE interface functions required to create ActiveDocuments. In fact, you can summarize the functionality of an ActiveDocument using what you've learned so far in the book as a basis for understanding. If you want your application to support this specification, it has to perform the four tasks listed below (including interface implementations):

- ✦ Implement IPersistStorage: Your application must support this interface and associated methods so that it can use OLE compound files as a storage medium. Of course, the client must also support the interface or it won't know to use your out-of-process server.
- ✦ Support OLE document embedding features: This particular feature is implemented in many different ways. The current trend is toward providing the user with two embedding methods through menu functions: Insert Object and Paste Special. The interfaces used to do this are IPersistFile, IOleObject, and IDataObject.
- ✦ Provide in-place activation support: There are two interfaces that you need to implement in order to support in-place activation: IOleInPlaceObject and IOleInPlaceActivateObject. To implement these interfaces, you have to gather information about the container using methods provided by the IOleInPlaceSite and IOleInPlaceFrame interfaces. In other words, ActiveDocument requires two-way communications between the container and the server.
- ✦ Add the ActiveDocument extensions: Most OLE 2 servers perform the first three tasks right now. To create an ActiveDocument application, you have to add the four interfaces discussed next.

Let's talk about the fourth item on this list in greater detail. ActiveDocuments are fairly new in some ways, although the support was part of Microsoft Office long before ActiveDocument even had a name. Visual C++ 4.2 provided marginal support for ActiveDocument, but the support for ActiveDocument in Visual C++ 6 is much better and allows you to develop a supporting application quickly.

Since Microsoft kept the specification for ActiveDocument a secret for so long, few compilers out there provide direct support for an ActiveDocument server. What this means to you, as a programmer, is that you have to either build the added interfaces yourself (not really too difficult) or upgrade your compiler. What you discover in the following sections are the four added interfaces that you have to support in order to create a fully functional ActiveDocument server. (Fortunately, they aren't all required. For example, Microsoft Excel implements only three of them.)



Microsoft Visual C++ versions 4.2 and above include direct support for creating most ActiveX object types, including ActiveDocument. If you have an older version of Visual C++, it's definitely time for an upgrade. The newer versions of Visual C++ literally build all of the elements I talk about in this section for you—reducing the time it takes for you to create your new control or application. If you absolutely must use an older version of Visual C++, you can find the ActiveDocument support you require in the ActiveX SDK. The SDK includes headers you need to add to your application to implement the required support. You also need to use the ActiveX SDK if you have another vendor's compiler that doesn't support ActiveX natively.

IOleDocument

Whenever an Internet client sees a server that implements the IOleDocument interface, it knows that the server can act as an ActiveDocument server. This is the first thing that Internet Explorer and other Internet client applications look for when they see a document associated with your application. Don't confuse standard, in-place activation with the kind used by an ActiveDocument server. You can build an application that supports in-place activation alone and works fine on a local machine, but it won't support in-place activation over the Internet.

In many cases, it doesn't matter if you implement this interface—you still can support OLE 2 without it. A server that's missing this interface can simply be opened in a separate window, even if it normally supports in-place activation with local clients such as Word for Windows.

So, what does this interface do besides telling Internet Explorer that your application supports ActiveDocuments? The methods it supports get called every time the client needs to create new server views (CreateView( ) method), enumerate those views (EnumViews( ) method), or retrieve the MiscStatus bits associated with the ActiveDocument (GetDocMiscStatus( ) method). In essence, this interface helps you manage the server as a whole. It provides the low-level functionality required for the client and server to communicate.

A server view isn't the same thing as the view that users see when they click a link to one of your documents — that's managed by the IOleDocumentView function discussed in the next section. What a server view provides is a single instance of the server itself. A client application uses this view for communication purposes. For example, if the client needs to find out what features the server supports, it uses a server view to do so. We'll see how this works when we get into the actual application code.

There are four standard miscellaneous status bits. The first bit, DOCMISC_ CANCREATEMULTIPLEVIEWS, tells the client whether the server can create multiple views. In other words, this bit defines whether you can run multiple copies of the application at one time. In most cases, a modern server can do this. About the only exceptions are for CAD or drawing programs where the memory requirements might be too prohibitive. Another class of application that may not support this is a communications program since most people only have one modem. The second bit, DOCMISC_SUPPORTCOMPLEXRECTANGLES, tells whether the server can support complex view-area commands. An example is whether or not the server will allow the client to determine the position of things such as scroll bars and sizing boxes. The third bit, DOCMISC_CANOPENEDIT, is used to tell the client whether the server can open a document for editing. Setting this bit prevents the user from editing a document online (something you may want to consider for security reasons, or if you want to create a server for viewing purposes only). The final bit, DOCMISC_NOFILESUPPORT, tells whether the server supports any kind of file manipulation. Setting this bit usually forces the client to display an error message since the user can't read the selected file.

IOleDocumentView

Like the IOleDocument interface, you must implement IOleDocumentView to make ActiveDocument work. This particular interface element relies on the IOleDocument interface. You need to have a server running before you can open a document. In addition, the client relies on information it gets from the GetDocMiscStatus( ) method of IOleDocument to know how to interact with this interface. Each copy of the IOleDocumentView interface controls a single instance of an ActiveDocument view. In most cases, this means that the single instance of the IOleDocumentView interface controls a single document. However, you can easily create one instance of the class for each view of the single document you open.

The IOleDocumentView interface supports a variety of methods. The following list contains the most common methods that you'll find. As you can see, the methods allow you to resize the screen or create another copy of the view you're looking at. Some of the other methods allow you to reset the view's bounding area or determine which document is currently displaying within the view.

SetInPlaceSite( )	Associates a view site object with this view. The client supplies the view site object. In essence, this method associates a document with the current view.
GetInPlaceSite( )	Returns a pointer to the view site object associated with the view.
GetDocument( )	Returns a pointer to the document associated with the view.
SetRect( )	Defines the bounding area for the view. In other words, this method sets the size of the window that the user sees.
GetRect( )	Returns the coordinates for the view's bounding area.
SetRectComplex( )	Defines a complex bounding area for the view. Not only does this method determine the size of the window the user sees, but also determines things such as the placement of scroll bars and other view elements. A view doesn't have to support this feature. You do need to set a miscellaneous status bit if your application doesn't provide the support. (See the IOleDocument interface description for more details.)
Show( )	The client can use this method to either show or hide the view.
UIActivate( )	Determines whether the user interface is active or not. Normally, the user interface is active only when the view has the focus. It's normally deactivated at all other times to prevent conflicts with the view that does have focus.

Open( )	Requests that the server open the view in a separate window. You can turn off this feature using a miscellaneous status bit. (See the IOleDocument interface description for more details.)
CloseView( )	Shuts the view down.
SaveViewState( )	Writes the current view status information to an IStream.
ApplyViewState( )	Requests that a view return its state to the settings defined in a previously saved IStream.
Clone( )	Creates a copy of the current view. The cloned view has the same context, but uses a different view port (instance of the IOleDocumentView interface).

IOleCommandTarget

This is one of the interfaces you don't have to implement to make ActiveDocument work. However, it's a lot more than a simple convenience item. IOleCommandTarget allows the client and server to talk with each other without resorting to tricks like assigning fixed-menu IDs. Of course, there are limitations to this communication. For one thing, the communication remains limited to a fixed number of commands (which I talk about a little further down). There's a two-step procedure required to make this part of the interface work.

The first part of the client-to-server communication is to find out what commands the server supports. The client does this using the Query( ) method. The following list provides a complete description of the commands that the server can support along with their associated identifiers. The first thing you should notice is that most of the commands are standard menu entries.

Edit Clear	OLECMDID_CLEARSELECTION
Edit Copy	OLECMDID_COPY
Edit Cut	OLECMDID_CUT
Edit Paste	OLECMDID_PASTE
Edit Paste Special	OLECMDID_PASTESPECIAL
Edit Redo	OLECMDID_REDO
Edit Select All	OLECMDID_SELECTALL
Edit Undo	OLECMDID_UNDO
File New	OLECMDID_NEW
File Open	OLECMDID_OPEN
File Page Setup	OLECMDID_PAGESETUP
File Print	OLECMDID_PRINT

File Print Preview	OLECMDID_PRINTPREVIEW
File Properties	OLECMDID_PROPERTIES
File Save	OLECMDID_SAVE
File Save As	OLECMDID_SAVEAS
File Save Copy As	OLECMDID_SAVECOPYAS
View Menu Commands	Not a standard command. This identifier asks the server if it can perform the following three tasks: return a zoom value, display a zoom dialog box, and set a zoom value. This identifier normally is associated with View menu commands (or their equivalent) if the server supports them. OLECMDID_ZOOM
View Zoom	Not a standard command. This identifier retrieves the zoom range supported by the server. It's normally associated with the View Zoom command if the server supports it (or an equivalent). OLECMDID_GETZOOMRANGE
Tools Spelling	OLECMDID_SPELL

The second phase of the client-to-server communication uses the Exec method. The client passes the server one or more OLECMD structures. Each structure contains a single command, any required input arguments, and a place to put informational flags on return from the call. You don't need to provide any input arguments, so that part of the structure contains a NULL. The standard options appear in the following list.

OLECMDEXECOPT_PROMPTUSER	Prompts the user for some kind of input prior to executing the command. For example, you want to use this option with a File Open command.
OLECMDEXECOPT_DONTPROMPTUSER	Don't ask the user for any kind of input. For example, you might want to use this option when the user asks you to print a document.
OLECMDEXECOPT_DODEFAULT	You're not sure whether to prompt the user or not. In this case, you want the application to perform the default action. In most cases, this means it prompts the user for input.

OLECMDEXECOPT_SHOWHELP	Don't execute the command at all. Display its help screen instead. You might want to use this command if your ActiveDocument provides an alternative Help button.

The exec method also returns a value so that you can determine the outcome of the client-to-server communication. It's important to check these return values to ensure the communication took place as anticipated before you perform any additional processing steps. The following list describes the flags that you see on return from an Exec call.

OLECMDF_SUPPORTED	The view object supports the requested command.
OLECMDF_ENABLED	The command is available and the view object has enabled it.
OLECMDF_LATCHED	This command uses an on-off toggle and it is currently set to on.
OLECMDF_NINCHED	The view object can't determine the state of a command that uses a toggle state. In most cases, this means that the command uses a tristate configuration and that it is in the indeterminate state. For example, a three-state checkbox returns this value if the user selects some suboptions for an install program but not others. (The checkbox appears grayed onscreen.)

IPrint

IPrint is another optional interface that you can implement. This interface allows an object to support programmatic printing. There are three methods supported by IPrint: print (Print), retrieve print-related information (GetPageInfo), and set the initial page number for a print job (SetInitialPageNum). Of the three methods, only the Print method accepts any flags as input. The following list provides a description of these flags and explains how to use them.

PRINTFLAG_MAYBOTHERUSER	Tells the server that user interaction is allowed by the client. If this flag isn't set, then any print requests have to run by themselves. In most cases, the client will allow user interaction — the only exception probably involves batch printing jobs or situations when the print operation proceeds in the background.

PRINTFLAG_PROMPTUSER	Prompts the user for input regarding the print job using the standard print dialog box (like the one supported by Windows). For example, the user can select the number of copies when this option is specified. You also must specify the PRINTFLAG _MAYBOTHERUSER flag to use this option.
PRINTFLAG_USERMAYCHANGEPRINTER	Allows the user to change the printer. There are some situations when you don't want to enable this option — such as network setups where the user can't easily access the printer. You also must specify the PRINTFLAG _PROMPTUSER flag to use this option.
PRINTFLAG_RECOMPOSETODEVICE	Tells the print job to recompose itself for the target printer. For example, if the target printer supports a higher resolution than currently specified as part of the print job, then the print job should make use of that higher resolution.
PRINTFLAG_DONTACTUALLYPRINT	Test the print job out, but don't actually create any output. This option allows you to test a user interface feature such as prompting without wasting paper in the process.
PRINTFLAG_PRINTTOFILE	Send the printed output to a file instead of a printer.

Summary

This chapter provides you with a lot of insights into the process of determining which kind of component technology to use, then implementing that technology in the form of a new component. I've concentrated our discussion on the kinds of components that you would use to create a desktop or end-user, Web-based application. There's a wealth of other component types that I discuss as the book

progresses. Here are some additional conclusions you can draw from the contents of this chapter.

- Even though ActiveX and COM+ are both based on COM, there are differences in the way you implement these two technologies. Determining which technology is best for a given situation may not be crystal clear, but you need to make the decision before you begin the design process.
- There's more than one flavor of ActiveX development technology to choose from today. The two most popular methodologies are MFC and ATL.
- The main advantage of using MFC is development speed. You can create a control relatively quickly using MFC with a lot less COM experience. In addition, MFC provides you with more automated development tools and the full MFC library for support.
- The main advantages of using ATL are control size, speed, and flexibility. You can create a variety of different control types that are fully optimized with regard to supported interfaces and features. In addition, ATL doesn't assume anything about the control you want to create, including the need for resources like property pages. As a result, an ATL control ends up being more efficient than its MFC counterpart.
- MFC-based ActiveX controls are best used on the desktop or on an intranet if you have a high-speed connection to the server.
- ATL-based ActiveX controls are best used on the Internet or on the desktop if development time isn't an issue.
- ActiveX isn't always the most appropriate technology to use on the Internet, especially if you can't be sure about the capabilities of the client browser. Other technologies such as ISAPI, scriptlets, or Java applets often work better on Web pages designed for general access by a diverse audience.
- The biggest difference between in-process and out-of-process servers is that an in-process server loads in the client address space, whereas an out-of-process server resides outside the client in its own process. In addition, an in-process server normally resides within a DLL, whereas an out-of-process server normally resides in an EXE file.
- ActiveDocument provides a method for a user to view documents within an application that doesn't normally support them. All ActiveDocument applications are implemented as out-of-process servers.
- The four essential ActiveDocument interfaces from a server perspective are IOleDocument, IOleDocumentView, IOleCommandTarget, and IPrint. Of these four interfaces, only IPrint is truly optional. Even though the IOleCommandTarget interface is optional, it's normally a good idea to implement it. Your server also needs to perform the other tasks mentioned in the "ActiveDocument Required Interfaces" section of this chapter.

✦ ✦ ✦